# Securing Distributed Systems with Information Flow Control

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières
*Stanford University*

## ABSTRACT

Recent operating systems [12, 21, 26] have shown that decentralized information flow control (DIFC) can secure applications built from mostly untrusted code. This paper extends DIFC to the network. We present DStar, a system that enforces the security requirements of mutually distrustful components through cryptography on the network and local OS protection mechanisms on each host. DStar does not require any fully-trusted processes or machines, and is carefully constructed to avoid covert channels inherent in its interface. We use DStar to build a three-tiered web server that mitigates the effects of untrustworthy applications and compromised machines.

## 1 INTRODUCTION

Software systems are plagued by security vulnerabilities in poorly-written application code. A particularly acute example is web applications, which are constructed for a specific web site and cannot benefit from the same level of peer review as more widely distributed software. Worse yet, a sizeable faction of web application code actually comes from third parties, in the form of libraries and utilities for tasks such as image conversion, data indexing, or manipulating PDF, XML, and other complex file formats. Indeed, faced with these kinds of tasks, most people start off searching for existing code. Sites such as FreshMeat and SourceForge make it easy to find and download a wide variety of freely available code, which unfortunately comes with no guarantee of correctness. Integrating third-party code is often the job of junior engineers, making it easy for useful but buggy third-party code to slip into production systems. It is no surprise we constantly hear of catastrophic security breaches that, for example, permit a rudimentary attacker to download 100,000 PayMaxx users' tax forms [20].

If we cannot improve the quality of software, an alternative is to design systems that remain secure despite untrustworthy code. Recent operating systems such as Asbestos [21], HiStar [26], and Flume [12] have shown this can be achieved through decentralized information flow control (DIFC). Consider again the PayMaxx example, which runs untrustworthy application code for each user to generate a tax form. In a DIFC operating system, we could sandwich each active user's tax form generation process between the payroll database and an HTTPS server, using the DIFC mechanism to prevent the application from communicating with any other component. Figure 1 illustrates this configuration. Suppose the HTTPS server forwards the username and password supplied by the browser to the database for verification, the database only allows the application to access records of the verified user, and the HTTPS server only sends the application's output to the same web browser that supplied the password. Then even a malicious application cannot inappropriately disclose users' data. Effectively we consider the application *untrusted*, mitigating the consequences if it turns out to be *untrustworthy*.

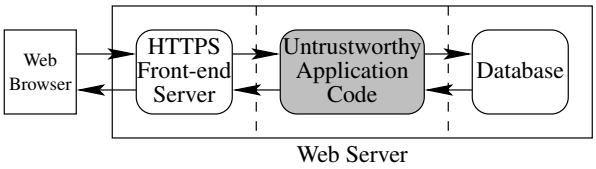



**Figure 1**: High-level structure of a web application. The shaded application code is typically the least trustworthy of all components, and should be treated as untrusted code to the extent possible. Dashed lines indicate how the web server can be decomposed into three kinds of machines for scalability—namely, front-end, application, and data servers.

While DIFC OSes allow us to tolerate untrustworthy applications, they can do so only if all processes are running on the same machine. Production web sites, on the other hand, must scale to many servers. A site like PayMaxx might use different pools of machines for front-end HTTPS servers, application servers, and back-end database servers. To achieve Figure 1's configuration when each component runs on a separate machine, we also need a network protocol that supports DIFC.

This paper presents DStar, a protocol and framework that leverages OS-level protection on individual machines to provide DIFC in a distributed system. At a high level, DStar controls which messages sent *to* a machine can affect which messages sent *from* the machine, thereby letting us plumb together secure systems out of untrusted components. DStar was designed to meet the following goals:

**Decentralized trust.** While an operating system has the luxury of an entirely trusted kernel, a distributed system might not have any fully-trusted code on any machine. Web applications rely on fully-trusted certificate authorities such as Verisign to determine what servers to trust: for example, Google's web applications might trust any server whose certificate from Verisign gives it a name ending in *.google.com*. However, we believe that requiring such centralized trust by design needlessly stifles innovation and reduces security. Instead, DStar strives to provide a more general, decentralized mechanism that lets applications either use authorities such as Verisign

by convention or manage their trust in other ways. Moreover, the lack of a single trusted authority simplifies integration of systems in different administrative domains.

**Egalitarian mechanisms.** One of the features that differentiates DIFC work from previous information flow control operating systems is that DIFC mechanisms are specifically designed for use by applications. Any code, no matter how unprivileged, can still use DIFC to protect its data or further subdivide permissions. This property greatly facilitates interaction between mutually distrustful components, one of the keys to preserving security in the face of untrustworthy code. By contrast, military systems, such as [22], have also long controlled information flow, but using mechanisms only available to privileged administrators, not application programmers. Similarly, administrators can already control information flow in the network using firewalls and VLANs or VPNs. DStar's goal is to offer such control to applications and to provide much finer granularity, so that, in the PayMaxx example, every user's tax information can be individually tracked and protected as it flows through the network.

**No inherent covert channels.** Information flow control systems inevitably allow some communication in violation of policy through *covert channels*. However, different applications have different sensitivities to covert channel bandwidth. PayMaxx could easily tolerate a 1,000-bit/second covert channel: even if a single instance of the untrusted application could read the entire database, it would need almost a day to leak 100,000 user records of 100 bytes each. (Exploiting covert channels often involves monopolizing resources such as the CPU or network, which over such a long period could easily be detected.) By contrast, military systems consider even 100 bits/second to be high bandwidth. To ensure that DStar can adapt to different security requirements, our goal is to avoid any covert channels inherent in its interface. This allows any covert channels to be mitigated in a particular deployment without breaking backwards compatibility. Avoiding inherent covert channels is particularly tricky with decentralized trust, as even seemingly innocuous actions such as querying a server for authorization or allocating memory to hold a certificate can inadvertently leak information.

DStar runs on a number of operating systems, including HiStar, Flume, and Linux. DStar leverages the OS's DIFC on the the former two, but must trust all software on Linux. Nonetheless, running DStar on Linux is convenient for incremental deployment; for example we can run the least-trusted application components on HiStar or Flume, and sandwich them between existing Linux web server and database machines.

To illustrate how DStar is used, Section 5.1 describes a secure web server we built on HiStar, and Section 5.3 shows how we distributed it using DStar. On a single machine, our web server ensures that the SSL certificate private key is accessible only to a small part of the SSL library, and can only be used for legitimate SSL negotiation. It also protects authentication tokens, such as passwords, so that even the bulk of the authentication code cannot disclose them. In a distributed setting, DStar allows the web server to ensure that private user data returned from a data server can only be written to an SSL connection over which the appropriate user has authenticated himself. DStar's decentralized model also ensures that even web server machines are minimally trusted: if any one machine is compromised, it can only subvert the security of users that use or had recently used it.

## 2 Information flow control

DStar enforces DIFC with labels. This section first describes how these labels—which are similar to those of DIFC OSes—can help secure a web application like PayMaxx. We then detail DStar's specific label mechanism.

### 2.1 Labels

DStar's job is to control how information flows between processes on different machines—in other words, to ensure that only processes that should communicate can do so. Communication permissions are specified with *labels*. Each process has a label; whether and in which direction two processes may communicate is a function of their labels. This function is actually a partial order, which we write $\sqsubseteq$ (pronounced "can flow to").

Roughly speaking, given processes $S$ and $R$ labeled $L_S$ and $L_R$, respectively, a message can flow from $S$ to $R$ only if $L_S \sqsubseteq L_R$. Bidirectional communication is permitted only if, in addition, $L_R \sqsubseteq L_S$. Labels can also be incomparable; if $L_S \not\sqsubseteq L_R$ and $L_R \not\sqsubseteq L_S$, then $S$ and $R$ may not communicate in either direction. Such label mechanisms are often referred to as "no read up, no write down," where "up" is the right hand side of $\sqsubseteq$. Given that labels can be incomparable, however, a more accurate description might be, "only read down, only write up."

Because a distributed system cannot simultaneously observe the labels of processes on different machines, DStar also labels messages. When $S$ sends a message $M$ to $R$, $S$ specifies a label $L_M$ for the message. The property enforced by DStar is that $L_S \sqsubseteq L_M \sqsubseteq L_R$. Intuitively, the message label ensures that untrusted code cannot inappropriately read or disclose data. In the PayMaxx example, the payroll database should use a different value of $L_M$ to protect the data of each user in the database, so that only the appropriate instance of the tax form generating application and HTTPS front ends can receive messages containing a particular user's data.

### 2.2 Downgrading privileges

If data could flow only to higher and higher labels, there would be no way to get anything out of the system: the

high label of the HTTPS front end would prevent it from sending tax forms back to a client web browser over a network device with a low label. The big difference between DIFC and more traditional information flow [8] is that DIFC *decentralizes* the privilege of bypassing "can flow to" restrictions. Each process $P$ has a set of privileges $O_P$ allowing it to omit particular restrictions when sending or receiving messages, but no process has blanket permission to do so, In effect, $O_P$ lets $P$ *downgrade* or lower labels on message data in certain ways.

We write $\sqsubseteq_{O_P}$ ("can flow to, given privileges $O_P$") to compare labels in light of a set of privileges $O_P$. As we will show later, $\sqsubseteq_{O_P}$ is strictly more permissive than $\sqsubseteq$; in other words, $L_1 \sqsubseteq L_2$ always implies $L_1 \sqsubseteq_{O_P} L_2$, but not vice versa. What DStar actually enforces, then, is that $S$ can send a message $M$ that $R$ receives only if

$$L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{O_R} L_R.$$

In other words, $S$ with privileges $O_S$ can produce a message labeled $L_M$ and $R$ with privileges $O_R$ can receive a message so labeled. In the PayMaxx example, one might give the untrusted tax form generator no privileges, while giving the HTTPS front end the ability to downgrade $L_M$ so as to send data back to the client web browser.

But who assigns such labels and privileges to the different processes in a system? Very often the security policy we want is a conjunction of concerns by mutually distrustful components. For example, the payroll database may want to disclose user records only after the the front-end server has supplied the appropriate password, while the front-end server may want to prevent the database from disclosing plaintext passwords sent to it for authentication. If DStar concentrated the ability to set labels in a group of specially-designated, privileged processes, it would impede the ability of unprivileged software to express security requirements. That, in turn, would lead to more trusted components and increased damage should any of them in fact be untrustworthy. Fortunately, as described in the next subsection, decentralized downgrading helps make DStar's labels an egalitarian mechanism with which any process can express security concerns.

## 2.3 Categories

A DStar label is a set of *categories*, each of which imposes a restriction on who can send or receive data. There are two types of category: *secrecy* and *integrity*. Secrecy categories in a message label restrict who can receive the message. In the PayMaxx example, to avoid improper disclosure, the database server should have one secrecy category for each user and include that category in the label of any messages containing that user's data. Conversely, integrity categories in a message label constrain who may have sent the message, and thus can help authenticate the sender. We will use $s$ and $i$ subscripts to indicate secrecy and integrity categories, respectively.

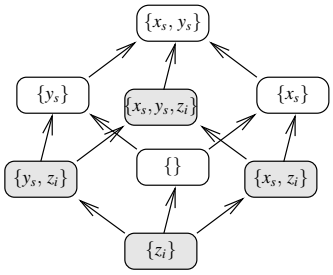



**Figure 2**: Lattice formed by labels using two secrecy categories, $x_s$ and $y_s$, and an integrity category, $z_i$. Shading indicates labels that include the $z_i$ category. Arrows show pairs of labels where the "can flow to" $\sqsubseteq$ relation holds, and thus how messages can be sent. Self-arrows are not shown. Information can also flow transitively over multiple arrows.

For any two labels $L_1$ and $L_2$, we can now formally define the $\sqsubseteq$ relation:

> $L_1 \sqsubseteq L_2$ if and only if $L_1$ contains all the integrity categories in $L_2$ and $L_2$ contains all the secrecy categories in $L_1$.

As illustrated in Figure 2, labels form a lattice under the $\sqsubseteq$ relation, and thus transitively enforce a form of mandatory access control [8].

A process $P$'s downgrading privileges, $O_P$, are also represented as a set of categories. We say $P$ *owns* a category $c$ when $c \in O_P$. Ownership confers the ability to ignore the restrictions imposed by a particular category at the owner's discretion. For any two labels $L_1$ and $L_2$ and privileges $O$, we can now formally define $\sqsubseteq_O$:

$$L_1 \sqsubseteq_O L_2 \text{ if and only if } L_1 - O \sqsubseteq L_2 - O.$$

In other words, except for the categories in $O$, $L_1$ contains all the integrity categories in $L_2$ and $L_2$ contains all the secrecy categories in $L_1$.

What makes categories egalitarian is that any process can allocate a category and simultaneously gain ownership of the newly allocated category. A process that owns a category may also, at its discretion, *grant* ownership of that category to another process. Every DStar message $M$ includes a set of categories $G_M$ that its sender $S$ is granting to the recipient, where DStar ensures that

$$G_M \subseteq O_S.$$

Thus, the payroll database can create one category per user and grant ownership of the category to the appropriate HTTPS front end, as illustrated in Figure 3. At the same time, the front end can allocate its own categories to protect plaintext passwords sent the to the database.

## 2.4 Clearance

In addition to its label $L_P$ and privileges $O_P$, each process $P$ has a third set of categories, $C_P$, called its *clearance*. Clearance represents the right of a process to raise its own label. A process may set its label to any value $L_P^{\text{new}}$

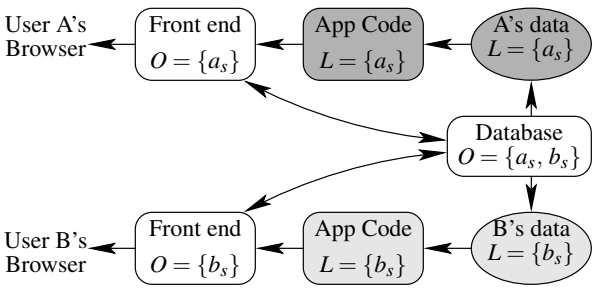



**Figure 3**: Example use of labels to prevent the PayMaxx application code from inappropriately disclosing data, with two users, A and B. Rounded boxes are processes. Ellipses are messages. Shaded components are labeled with a user's secrecy category, $a_s$ or $b_s$ for $A$ and $B$ respectively. The front end communicates with the database to authenticate the user and obtain ownership of the user's secrecy category.

such that $L_P \sqsubseteq L_P^{\text{new}} \sqsubseteq C_P$. A process that owns a category can raise the clearance of other processes in that category. For instance, when the front-end server sends a user's password to the payroll database, the database responds by granting it ownership of the user's secrecy category. Given this ownership, the front end can raise the clearance of the application, which in turn allows the application to add that user's secrecy category to its label. DStar fully implements clearance, but for simplicity of exposition we mostly omit it from discussion.

## 3 DStar exporter

To model the fact that all processes with direct access to a particular network can exchange messages, DStar requires all such processes to have the same label, $L_{\text{net}}$, which we call the *network label*. (For networks connected to the Internet, $L_{\text{net}} = \{\}$.) However, our main goal of restricting the communication of potentially untrustworthy code requires running different processes with different labels. In order to allow these processes, which lack direct network access, to communicate across machines, DStar introduces the notion of *exporter* daemons.

Each host runs an exporter daemon, which is the only process that sends and receives DStar messages directly over the network. When two processes on different machines communicate over DStar, they exchange messages through their local exporters, as shown in Figure 4. Exporters are responsible for enforcing the information flow restrictions implied by message labels, which boils down to three requirements. Each exporter $E$ must:

1. Track the labels assigned to, and the categories owned by, every process on $E$'s machine,
2. Ensure that processes on $E$'s machine cannot send or receive messages with inappropriate labels, and
3. Send or accept a network message $M$ only when $E$ can trust the remote exporter to respect the communication restrictions implied by $M$'s label.

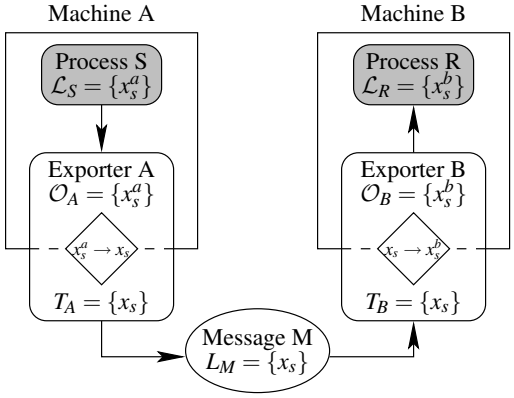



**Figure 4**: Message sent by process $S$ on machine $A$ to process $R$ on machine $B$. $\mathcal{L}$ and $\mathcal{O}$ designate local OS labels and OS ownership privileges. $x_s^a$ and $x_s^b$ are local OS secrecy categories. Exporters translate between these OS categories and the globally-meaningful DStar category $x_s$. $T_A$ is the set of DStar categories whose owners explicitly trust exporter $A$, while $T_B$ is the analogous set for exporter $B$.

The next three subsections discuss each of these requirements in turn.

### 3.1 Local OS DIFC

To track the labels and ownership privileges of local processes, exporters rely on the DIFC facilities of the local operating system. Asbestos, HiStar, and Flume all have mechanisms similar to DStar's categories and labels. We will always refer to these as *OS categories* and *OS labels* to differentiate them from DStar's categories and labels. Every exporter establishes mappings between OS categories, which have meaning only on the local machine, and DStar categories, which are globally meaningful. At a high level, one can view the exporter's job as translating between local OS DIFC protection of data on the host and cryptographic protection of data on the network. Because an exporter's label is always the network label, the exporter must own all local OS categories required to access the messages it processes.

Not every OS category has an equivalent DStar category. However, every category in a process $P$'s DStar label, $L_P$, is represented by a corresponding OS category in $P$'s OS label, $\mathcal{L}_P$. Each restriction implied by a DStar category is thus locally enforced by the corresponding OS category, thereby ensuring processes on the same machine cannot use local OS facilities to communicate in violation of their DStar labels. Similarly, every DStar category that $P$ owns is reflected in $P$'s local OS downgrading privileges, $\mathcal{O}_P$. Processes must explicitly invoke the exporter to create a mapping between a local and a DStar category. Section 4.2 details this process for HiStar. An important point is that the exporter creates unforgeable mappings, but does not store them. Processes must arrange to store the mappings they care about and specify them when sending messages.

### 3.2 Local exporter checks

Recall from Section 2.2 that when a process $S$ sends a message $M$ to a process $R$ on another machine, DStar must ensure $L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{O_R} L_R$. If $M$ grants ownership privileges, DStar must also ensure $G_M \subseteq O_S$.

The left half of the first requirement, $L_S \sqsubseteq_{O_S} L_M$, and the second requirement, $G_M \subseteq O_S$, are enforced on $S$'s machine. Roughly speaking, the local OS ensures that these relations hold in terms of local categories, and the exporter translates local categories into DStar categories. In particular, when $S$ sends $M$ to $S$'s exporter through the local OS's IPC mechanism, it uses the OS's DIFC both to label the message $\mathcal{L}_M$ (requiring $\mathcal{L}_S \sqsubseteq_{\mathcal{O}_S} \mathcal{L}_M$) and to prove to the exporter that it owns $\mathcal{G}_M$ (requiring $\mathcal{G}_M \subseteq \mathcal{O}_S$). $S$'s exporter, in turn, uses mappings it previously created to translate the OS categories in $\mathcal{L}_M$ and $\mathcal{G}_M$ into DStar categories to produce $L_M$ and $G_M$.

The right half of DStar's first requirement, $L_M \sqsubseteq_{O_R} L_R$, is enforced on $R$'s machine in an analogous fashion. Upon receiving $M$ from the network, $R$'s exporter translates the DStar categories in $L_M$ and $G_M$ into OS categories to obtain a local OS message label $\mathcal{L}_M$ and set of OS categories $\mathcal{G}_M$ that should be granted to $R$. (It performs this translation using mappings it previously created, which must be explicitly referenced in the message.) The exporter then sends $M$ through the local OS's IPC mechanism to $R$ with a label of $\mathcal{L}_M$, and grants $R$ ownership of categories in $\mathcal{G}_M$. The local OS DIFC mechanism, in turn, ensures that $\mathcal{L}_M \sqsubseteq_{\mathcal{O}_R} \mathcal{L}_R$.

Together, the two exporters guarantee $L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{O_R} L_R$, but only if they are both trustworthy. The final task of an exporter is to determine whether or not it can trust the remote exporter to enforce its half of the equation.

### 3.3 Decentralized trust

When should an exporter trust another exporter to send or receive a message $M$? In order to support decentralized trust, DStar leaves this decision up to the owners of the categories whose security is at stake—namely those in which $L_M$ differs from $L_{\text{net}}$, and those in $G_M$. Each exporter $E$ has a set of categories $T_E$ whose owners trust $E$ to handle the category. We refer to $T_E$ as $E$'s *trust set*.

When a process $P$ creates a DStar category $c$, $P$ implicitly adds $c$ to its local exporter's trust set. Before $P$ can either grant ownership of $c$ to a remote process $Q$ or raise $Q$'s clearance in $c$, $P$ must explicitly add $c$ to the trust set of $Q$'s exporter. This reflects the reality that a process can only be as trustworthy as its exporter.

Adding category $c$ to an exporter $E$'s trust set has the same security implications as granting ownership of $c$ to a process. The difference is that while an exporter has no way of verifying the local OS ownership privileges of a process on another machine, it *can* verify the trust set of a remote exporter.

Consider $S$ on machine $A$ sending $M$ to $R$ on machine $B$, as in Figure 4. Treating trust as ownership, we can view this as two message exchanges. From exporter $A$'s point of view, $M$ flows from $S$ to exporter $B$. Therefore, DStar should ensure that $L_S \sqsubseteq_{O_S} L_M \sqsubseteq_{T_B} L_B$ and $G_M \subseteq O_S$. From exporter $B$'s point of view, the message flows from exporter $A$ to $R$, which requires $L_A \sqsubseteq_{T_A} L_M \sqsubseteq_{O_R} L_R$ and $G_M \subseteq T_A$. Because they directly access the network, both exporters have the same label, $L_A = L_B = L_{\text{net}}$. It therefore remains to verify each other's trust sets.

DStar implements trust sets using certificates. Each exporter has a public/private key pair. If an exporter is trusted in a particular category, it can use its private key to sign a certificate delegating trust in that category to another exporter, named by its public key. Certificates include an expiration time to simplify revocation.

To avoid any trusted, central naming authority, DStar uses *self-certifying* category names that include the public key of the exporter that created the category. An exporter is trusted in every category it creates by definition. Given a category's name, other exporters can verify certificates signed by its creator using the public key in the name. Further delegation of trust requires the entire chain of certificates leading to the category's creator.

In many cases, even contacting another machine over the network to query it for credentials can inappropriately leak information. Thus, an important property of DStar certificates is that they allow one exporter to verify membership of a category in a remote exporter's trust set with no external communication.

### 3.4 Addressing

Delegation certificates determine when an exporter with a particular public key can receive a message with a particular message label. However, at a low level, network messages are sent to network addresses, not public keys. Any communication to map exporter public keys to network addresses could in itself leak information. We therefore introduce *address certificates*, which contain the exporter's current IP address signed by the exporter's key. Exporters only send messages to other exporters for which they have address certificates.

Unfortunately, most networks cannot guarantee that packets sent to an IP address will only reach the intended recipient. On a typical Ethernet, an active attacker can spoof ARP replies, overflow MAC address tables, or flood the network to gain information about communication patterns between other hosts. While DStar encrypts and MACs its network traffic, the mere presence and destination of encrypted packets may communicate information. Malicious code on a HiStar machine could, for instance, exploit this fact to leak information to a colluding Linux box on the same network. While the present level of trust in the network suffices for many applications,

in the future we intend to integrate DStar with network switches that can better conceal communication [6].

Exporters currently distribute address certificates by periodically broadcasting them to the local-area network. Certificate expiration times allow IP address reuse. After expiration, other exporters will not connect to the old address. In a complex network, broadcast would not suffice to distribute address certificates to all exporters; one might need a partially-trusted directory service.

### 3.5 Exporter interface

Exporters provide unreliable one-way message delivery to communication endpoints called *slots*, which are analogous to message ports. Communication is unreliable to avoid potential covert channels through message acknowledgments; for instance, if process $S$ can send messages to $R$ but not vice-versa, a message delivery status could allow $R$ to convey information to $S$ by either accepting or refusing messages. However, in the common case where labels permit bi-directional communication, library code outside the exporter provides higher-level abstractions such as RPC, much the way RPC can be layered on top of unreliable transports such as UDP and IP.

We will now describe the DStar network protocol and exporter interface, starting with DStar's self-certifying category names:

```
struct category_name {
    pubkey creator;
    category_type type;
    uint64_t id;
};
```

Here, `type` specifies whether this is a secrecy or integrity category, and `id` is an opaque identifier used to distinguish multiple categories created by the same exporter. Exporters can create categories by picking a previously unused pseudo-random `id` value, for example by encrypting a counter with a block cipher.

Next, the format of all DStar messages is as follows:

```
struct dstar_message {
    pubkey recipient_exporter;
    slot recipient_slot;
    category_set label, ownership, clearance;
    cert_set certs;
    mapping_set mapset;
    opaque payload;
};
```

The message is addressed to slot `recipient_slot` on `recipient_exporter`'s machine. The `label` specifies information flow restrictions on the contents of the message, and consists of a set of `category_name`s as defined earlier. The recipient exporter will grant ownership and clearance of categories specified in `ownership` and `clearance` respectively to the recipient slot when it delivers the message.

`certs` contains delegation certificates proving to the recipient exporter that the sender is trusted with all of the categories in `ownership` and `clearance` (stated as $G_M \subseteq T_{\text{sender}}$ in Section 3.3), and, assuming $L_{\text{sender}} = L_{\text{net}} = \{\}$, trusted with all *integrity* categories in `label` (stated as $L_{\text{sender}} \sqsubseteq_{T_{\text{sender}}} L_M$ in Section 3.3). `mapset` contains mappings for the recipient exporter to map the necessary DStar categories to OS categories; we discuss these mappings in more detail in Section 4.2.

Each exporter provides to other processes on the same machine a single function to send DStar messages:

```
void dstar_send(ip_addr, tcp_port, dstar_message,
                cert_set, mapping_set);
```

Here, the `cert_set` and `mapping_set` have the opposite roles from those in `dstar_message`. They prove to the *sending* exporter that it is safe to send the supplied `dstar_message` to the recipient exporter. In particular, assuming $L_{\text{recipient}} = L_{\text{net}} = \{\}$, `cert_set` contains delegation certificates proving the recipient exporter is trusted in all *secrecy* categories in the message label (stated as $L_M \sqsubseteq_{T_{\text{recipient}}} L_{\text{recipient}}$ in Section 3.3), while `mapping_set` provides mappings allowing the sending exporter translate the necessary OS categories to DStar categories. `cert_set` must also include an address certificate proving that the given IP address and TCP port number reach the recipient exporter.

Finally, delivery of messages by an exporter to local processes will be discussed in Section 4.3.

### 3.6 Management services

DStar exporters provide additional functionality for management and bootstrapping, implemented as RPC servers on well-known slots. We will later illustrate how they are used in an application.

The **delegation service** allows a process that owns a category in the local operating system to delegate trust of the corresponding DStar category to another exporter, named by a public key. A signed delegation certificate is returned to the caller. This service is fully trusted; a compromise would allow an attacker to create arbitrary delegations and gain full control over all data handled by an exporter.

The **mapping service** creates mappings between DStar categories and local operating system security mechanisms; it will be discussed in more detail in Section 4.2. This service is also fully trusted.

The **guarded invocation service** launches executables with specified arguments and privileges, as long as a cryptographic checksum of the executable matches the checksum provided by the caller. The caller must have access to memory and CPU resources on the remote machine in order to run a process. This service is used in bootstrapping, when only the public key of a trusted exporter is known; the full bootstrapping process will be

described in more detail later on. This service is not trusted by the exporter—in other words, its compromise cannot violate the exporter's security guarantees. However, the exporter reserves a special slot name for this service, so that clients can contact this service on a remote machine during bootstrapping.

Finally, exporters provide a **resource allocation service**, which allows allocating resources, such as space for data and CPU time for threads, with a specified label on a remote machine. This service is used by the client-side RPC library to provide the server with space for handling the request; a timeout can be specified for each allocation, which allows garbage-collecting ephemeral RPC call state in case of a timeout. In the PayMaxx example, the front-end server uses this service to allocate memory and CPU time for running the application code on the application server. The system administrator also uses this service to bootstrap new DStar machines into an existing pool of servers, as will be described in Section 5.6. Although we describe this service here because it is used by a number of applications, it is not strictly necessary for an exporter to provide this service: applications could invoke the resource allocation service by using guarded invocation.

# 4 HiStar exporter

DStar exporters use the local operating system to provide security guarantees, and this section describes how exporters use HiStar and its category and label mechanism for this purpose. DStar also runs on Flume; the main difference there is that Flume does not explicitly label all memory resources, which can lead to covert channels or denial-of-service attacks.

To reduce the effect of any compromise, the HiStar exporter has *no superuser privileges*. The exporter runs as an ordinary process on HiStar without any special privileges from the kernel. The owner of each local category can explicitly allow the exporter to translate between that category on the local machine and encrypted DStar messages, by granting ownership of the local HiStar category to the exporter. The exporter uses this ownership to allow threads labeled with that category, which may not be able to send or receive network messages directly, to send and receive appropriately-labeled DStar messages.

To avoid covert channels, the HiStar exporter is largely stateless, keeping no per-message or per-category state. Instead, the exporter requires users to explicitly provide any state needed to process each message.

## 4.1 HiStar review

The HiStar kernel interface is designed around a small number of object types, including *segments*, *address spaces*, *containers*, *threads*, and *gates*. Like DStar, HiStar uses the notion of categories (implemented as opaque 61-bit values in the kernel) to specify information flow restrictions, and each object has a unique 61-bit object ID, and a label used for access control by the kernel, similar to a DStar label. For simplicity, this paper will use DStar labels to describe the labels of HiStar objects, although in reality an equivalent HiStar label is used, in which secrecy categories map to level **3**, integrity categories map to level **0**, ownership of categories maps to level $\star$, and the default level is **1**.

The simplest object type, a *segment*, consists of zero or more memory pages. An *address space* consists of a set of $VirtualAddress \rightarrow SegmentID$ mappings that define the layout of a virtual address space.

*Containers* are similar to directories, and all objects must exist in some container to avoid garbage collection. The root container is the only object in the system that does not have a parent container. Containers provide all resources in HiStar, including storage (both in memory and on disk) and CPU time.

*Threads* execute code, and consist of a register set and an object ID of an address space object that defines the virtual address space. Threads also have privileges—a set of categories owned by the thread. Any thread can allocate a fresh category, at which point it becomes the only one in the system with ownership of the new category.

*Gates* are used for IPC and privilege transfer. Unlike a typical IPC message port, gates require the client to donate initial resources—the thread object—for execution in the server's address space. Like threads, gates have privileges, which can be used when the thread switches to the server's address space. Each gate has an *access label* that controls who can invoke the gate: a thread $T$ can invoke gate $G$ with access label $A_G$ only if $L_T \sqsubseteq_{O_T} A_G$.

## 4.2 Category mappings

One of the main tasks of the exporter is to translate between the global space of DStar categories and the corresponding HiStar categories on the local machine. Since the exporter must be stateless, it is up to the users to supply these mappings for each message. However, these mappings are crucial to the security guarantees provided by the exporter—by corrupting these mappings, an attacker could convince the exporter to label an incoming secret message with a category owned by the attacker on the local machine, violating all security guarantees.

In the network protocol, exporters use signed certificates to get around this problem: users supply certificates to send each message, but exporters verify the signature on each certificate. However, on the local machine exporters also need ownership of the local HiStar category in order to be able to manipulate data labeled with that category. Since the HiStar kernel only allows category ownership to be stored in thread or gate objects, the ex-

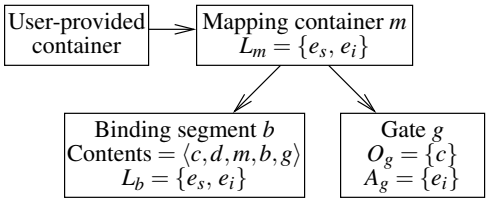



**Figure 5**: Objects comprising a mapping between DStar category $d$ and local HiStar category $c$. Arrows indicate that an object is in a container.

porter fundamentally requires memory (for a kernel object) for each category it handles on the local machine.

Thus, for each mapping between a DStar category and a HiStar category, the exporter needs two things: a kernel object storing ownership of the local HiStar category, and a secure binding between the DStar and HiStar category names. The secure binding could be represented by a certificate, but since the exporter already needs a kernel object to store ownership, we store the secure binding along with that kernel object, and avoid the overhead of public key cryptography.

HiStar's exporter represents each mapping using the objects shown in Figure 5. Container $m$ stores all other mapping objects, and in turn lives in a user-provided container, which allows the exporter itself to remain stateless. Gate $g$ stores the exporter's ownership of the local HiStar category. Finally, binding segment $b$ ensures that the user cannot tamper with the mapping despite providing the memory for the kernel objects, as follows.

The exporter owns two HiStar categories, $e_s$ and $e_i$, which it uses to ensure the security of *all* mappings on its machine. All objects comprising a mapping are labeled $\{e_s, e_i\}$, which ensures that only the exporter can create or modify them. The binding segment provides a secure binding between the DStar and HiStar category names, and contains the *mapping tuple*, $\langle c,d,m,b,g \rangle$. Users provide this tuple when they want to use a particular category mapping; the `mapping_set`, mentioned earlier in the `dstar_message` and the `dstar_send()` function, is a set of such tuples. The exporter verifies each tuple's integrity by checking that the tuple matches the contents of the binding segment, and that the binding segment has label $\{e_i, e_s\}$, so that only the exporter could have created it. Finally, to ensure that only the exporter can gain ownership of the HiStar category through the mapping gate, the gate's access label is set to $A_g = \{e_i\}$.

The mapping service, briefly mentioned earlier, allows applications to create new mappings. This service allows anyone to allocate either a fresh HiStar category for an existing DStar category, or a fresh DStar category for an existing HiStar category. Since the exporter is stateless, the caller must provide a container to store the new mapping, and grant the mapping service any privileges needed to access this container. The exporter does not grant ownership of the freshly-allocated category to the caller, making it safe for anyone to create fresh mappings. If the calling process does not own the existing category, it will not own the new category either, and will not be able to change the security policy set by the owner of the existing category. If the calling process does own the existing category, it can separately gain ownership of the new category, by sending a message that includes the existing category in the `ownership` field.

The mapping service also allows creating a mapping between an existing pair of HiStar and DStar categories, which requires the caller to prove ownership of both categories, by granting them to the mapping service.

### 4.3 Exporter interface

Exporters running on HiStar support two ways of communicating with other processes running on the same machine: *segments* and *gates*. Communicating via a segment resembles shared memory: it involves writing the message to the segment and using a futex [10] to wake up processes waiting for a message in that segment. Communication over a gate involves writing the message to a new segment, and then allocating a new thread, which in turn invokes the gate, passing the object ID of the message segment. Gates incur higher overhead for sending a message than segments, but allow passing ownership and clearance privileges when the thread invokes the gate.

As mentioned earlier, messages are delivered to slots, which in the case of HiStar names either a segment (by its object ID), or a gate (by its object ID and the object ID of a container to hold hold the newly-created message segment and thread). Exporters enforce labels of incoming messages by translating DStar labels into local HiStar labels, and making sure that the label of the slot matches the label of the message. Similarly, when a process sends a message, exporters ensure that the message label is between the label and clearance of the sending process.

The local exporter saves all address certificates it receives via broadcast from other exporters to a well-known file. This makes it easy for other processes to find address certificates for nearby exporters.

### 4.4 Implementation

The exporter comprises about 3,700 lines of C++ source code, and runs on HiStar, Flume, and Linux (though on Linux, all software must be trusted to obey information flow restrictions). The client library, trusted by individual processes to talk to the exporter, is 1,500 lines of C and C++ code. The exporter uses the libasync event-driven library [13] for network I/O and cryptography, and libc and libstdc++, which dwarf it in terms of code size.

## 5 APPLICATIONS

To illustrate how DStar helps build secure distributed systems, we focus on two scenarios that expand on our

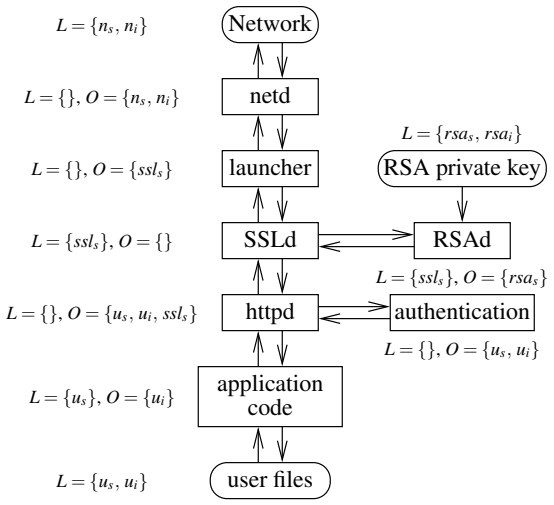



**Figure 6**: Architecture of the HiStar SSL web server. Rectangles represent processes. Rounded boxes represent devices and files.

running PayMaxx example. First, we describe the architecture of a highly privilege-separated web server we have built on HiStar, which partitions its privileges among many separate components to limit the effect of any single compromise, and discuss the security properties it achieves on one machine. We then show how this web server can be distributed over multiple machines like a typical three-tiered web application, providing performance scalability with similar security guarantees as on a single HiStar machine. Finally, we show how, even in an existing web service environment, DStar can be used to improve security by incrementally adding information flow control for untrusted code.

## 5.1 Web server on HiStar

Figure 6 shows the overall architecture of our privilege-separated SSL web server. The web server is built from a number of mutually-distrustful components to reduce the effects of the compromise of any single component. We first describe how requests are handled in this web server, and the next subsection will describe its security.

The TCP/IP stack in HiStar is implemented by a user-space process called ***netd***, which has direct access to the kernel network device. *netd* provides a traditional sockets interface to other applications on the system, which is used by our web server to access the network.

User connections are initially handled by the ***launcher***, which accepts incoming connections from web browsers and starts the necessary processes to handle them. For each request, the *launcher* spawns ***SSLd*** to handle the SSL connection with the user's web browser, and ***httpd*** to process the user's plaintext HTTP request. The *launcher* then proxies data between *SSLd* and the TCP connection to the user's browser. *SSLd*, in turn, uses the ***RSAd*** daemon to establish an SSL session key with the user's web browser, by generating an RSA signature using the SSL certificate private key kept by *RSAd*.

*httpd* receives the user's decrypted HTTP request from *SSLd* and extracts the user's password and request path from it. It then authenticates the user, by sending the user's password to that user's ***password checking agent*** from the HiStar authentication service [26]. If the authentication succeeds, *httpd* receives ownership of the user's secrecy and integrity categories, $u_s$ and $u_i$, and executes the ***application code*** with the user's privileges (in our case, we run GNU ghostscript to generate a PDF document). Application output is sent by *httpd* back to the user's web browser, via *SSLd* for encryption.

## 5.2 Web server security

The HiStar web server architecture has no hierarchy of privileges, and no fully trusted components; instead, most components are mutually distrustful, and the effects of a compromise are typically limited to one user, usually the attacker himself. Figure 7 summarizes the security properties of this web server, including the complexity of different components and effects of compromise.

The largest components in the web server, *SSLd* and the application code, are minimally trusted, and cannot disclose one user's private data to another user, even if they are malicious. The application code is confined by the user's secrecy category, $u_s$, and it is *httpd*'s job to ensure that the application code is labeled with $u_s$ when *httpd* runs it. Although the application code owns the user's integrity category, $u_i$, this only gives it the privilege to write to that user's files, and not to export them. Ownership of $u_i$ is necessary to allow the application code to read data not labeled with $u_i$, such as shared libraries. If the application code were to be labeled with $u_i$ instead, it would be restricted to reading only data labeled $u_i$, which would likely exclude needed binaries, shared libraries, and configuration files.

*SSLd* is confined by $ssl_s$, a fresh secrecy category allocated by the *launcher* for each new connection. Both the *launcher* and *httpd* own $ssl_s$, allowing them to freely handle encrypted and decrypted SSL data, respectively. However, *SSLd* can only communicate with *httpd* and, via the *launcher*, with the user's web browser.

*SSLd* is also not trusted to handle the SSL certificate private key. Instead, a separate and much smaller daemon, *RSAd*, has access to the private key, and only provides an interface to generate RSA signatures for SSL session key establishment. Not shown in the diagram is a category owned by *SSLd* that allows it and only it to invoke *RSAd*. Although a compromised *RSAd* can expose the server's SSL private key, it cannot directly compromise the privacy of user data, because *RSAd* runs confined with each user connection's $ssl_s$ category.

| Component | Lines of Code | Label | Ownership | Effects of Compromise |
|---|---|---|---|---|
| netd | 350,000 | $\{\}$ | $\{n_s, n_i\}$ | Equivalent to an active network attacker; subject to same kernel label checks as any other process |
| launcher | 310 | $\{\}$ | $\{ssl_s\}$ | Obtain plaintext requests, including passwords, and subsequently corrupt user data |
| SSLd | 340,000 | $\{ssl_s\}$ | $\{\}$ | Corrupt request or response, or send unencrypted data to same user's browser |
| RSAd | 4,600 | $\{ssl_s\}$ | $\{rsa_s\}$ | Disclose the server's SSL certificate private key |
| httpd | 300 | $\{\}$ | $\{u_s, u_i, ssl_s\}$ | Full access to data in attacker's account, but not to other users' data |
| authentication | 320 | $\{\}$ | $\{u_s, u_i\}$ | Full access to data of the user whose agent is compromised, but no password disclosure |
| application | 680,000+ | $\{u_s\}$ | $\{u_i\}$ | Send garbage (but only to same user's browser), corrupt user data (for write requests) |
| DStar exporter | 3,700 | | | Corrupt or disclose any data sent or received via DStar on a machine |
| DStar client library | 1,500 | | | Corrupt, but not necessarily disclose, data sent or received via DStar by an application |

**Figure 7**: Components of the HiStar web server, their complexity measured in lines of C code (not including libraries such as libc), their label and ownership, and the worst-case results of the component being compromised. The netd TCP/IP stack is a modified Linux kernel; HiStar also supports the lwIP TCP/IP stack, consisting of 35,000 lines of code, which has lower performance. The DStar exporter and client library illustrate the additional code that must be trusted in order to distribute this web server over multiple machines.

Side-channel attacks, such as [1], might allow recovery of the private key; OpenSSL uses RSA blinding to defeat timing attacks such as [5]. To prevent an attacker from observing intermediate states of CPU caches while handling the private key, *RSAd* starts RSA operations at the beginning of a 10 msec scheduler quantum (each 1024-bit RSA operation takes 1 msec), and flushes CPU caches when context switching to or from *RSAd* (with kernel support), at a minimal cost to overall performance.

The HiStar authentication service used by *httpd* to authenticate users is described in detail in [26], but briefly, there is no code executing with every user's privilege, and the supplied password cannot be leaked even if the password checker is malicious.

In our current prototype, *httpd* always grants ownership of $u_i$ to the application code, giving it write access to user data. It may be better to grant $u_i$ only to code that performs read-write requests, to avoid user data corruption by buggy read-only request handling code.

Our web server does not use SSL client authentication in *SSLd*. Doing so would require either trusting all of *SSLd* to authenticate all users, or extracting the client authentication code into a separate, smaller trusted component. In comparison, the password checking agent in the HiStar authentication service is 320 lines of code.

One caveat of our prototype is its lack of SSL session caching. Because a separate instance of *SSLd* is used for each client request, clients cannot reuse existing session keys when connecting multiple times, requiring public key cryptography to establish a new session key. This limitation can be addressed by adding a trusted SSL session cache that runs in a different, persistent process, at the cost of increasing the amount of trusted code.

### 5.3 Distributed web server

We have taken the HiStar web server described above, and used DStar to turn it into a three-tiered web application, as shown in Figure 8. HTTPS front-end servers run components responsible for accepting client connections and handling the HTTP protocol: the *launcher*, *SSLd*, *RSAd*, and *httpd*. Application servers run application code to execute requests. Finally, user data servers store private user data and perform user authentication.

HTTPS and application servers are largely stateless, making it easy to improve overall performance by adding more physical machines. This is an important consideration for complex web applications, where simple tasks such as generating a PDF document can easily consume 100 milliseconds of CPU time (using GNU ghostscript). User data servers can also be partitioned over multiple machines, by keeping a consistent mapping from each individual user to the particular user data server responsible for their data. Our prototype has a statically-configured mapping from users to user data servers, replicated on each HTTPS front-end server.

Our distributed web server has no central authority, and all data servers are mutually distrustful. For a web service that generates tax forms, this allows multiple companies to each provide their own data server. While each company may trust the web service to generate one employee's tax form, no company trusts anyone other than themselves with all of their employee data.

Similarly, different parts of a single user's data can be handled by different front-end and application servers. For example, if payment were required for accessing a tax document, separate front-end and application servers could be used to process credit card transactions. Even if those servers were to be compromised, user tax information would be handled by different front-end and application servers, and could remain secure. Conversely, if the servers handling tax data were compromised, credit card data would not necessarily be disclosed.

Interactions between components on a single machine are unchanged, with the same security properties as before. Interactions between components on different machines, on the other hand, maintain the same structure and security properties as before, but now go through the exporters on the respective machines, thereby making the exporters a part of the trusted code base. Communicating over DStar typically requires three things for each category involved: a mapping from the local category to a DStar category, certificates proving that the remote exporter is trusted by that DStar category, and a mapping from the DStar category to a local category on the remote machine, as we will describe next.

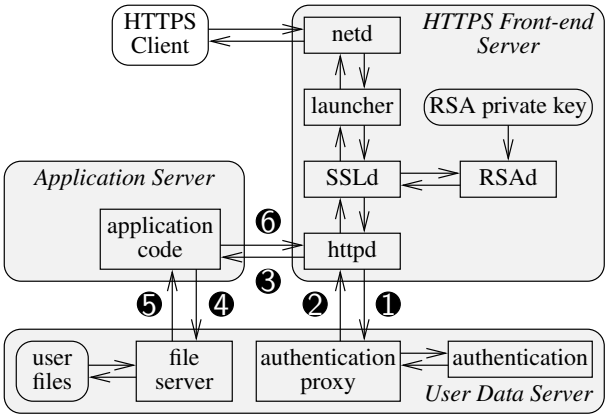



**Figure 8**: Structure of our web server running on multiple HiStar machines. Shaded boxes represent physical machines. Circled numbers indicate the order in which DStar messages are sent between machines. Not shown are DStar messages to create mappings on remote machines. Components have largely the same labels as in Figure 6.

The one exception where the distributed web server structurally differs from the one that runs on a single machine is authentication. We have not ported the three-phase HiStar authentication service to run over DStar yet; instead, we introduced a trusted authentication proxy to invoke the HiStar authentication service locally on the data server. *httpd* trusts the authentication proxy to keep the user's password secure, but guarded invocation ensures that the user's password is only passed to the correct authentication proxy process.

These changes to the HiStar web server added 740 lines of C++ code: 280 lines to *httpd*, a 140-line trusted authentication proxy, a 220-line untrusted RPC server to launch application code, and a 100-line file server.

To use DStar, applications, such as our distributed web server, must explicitly manage two important aspects of the distributed system. First, applications must explicitly define trust between the different machines in the distributed system, by creating and distributing the appropriate delegation certificates. Second, applications need to explicitly allocate resources, such as containers and category mappings, on different machines to be able to communicate between them and execute code remotely. The next two subsections describe how the distributed web server addresses these issues.

## 5.4 Trust management

The key trust relation in our distributed web server concerns the individual users' secrecy and integrity categories, or in other words, which machines are authorized to act on behalf of which users. All user categories in our design are initially created by, and therefore trust, the exporter on that user's data server. When the authentication proxy receives the correct user password (Step 1 in Figure 8), it asks the local exporter to create a short-lived delegation certificate, valid only for a few minutes, for the user's secrecy and integrity categories to the exporter on *httpd*'s front-end machine. Short-lived certificates ensure that, even if some machines are compromised, they can only subvert the security of users that are currently using, or have recently used those machines. The authentication proxy sends these certificates to *httpd* in Step 2, and grants it ownership of the user's categories using the `ownership` field of the message.

Although *httpd* receives ownership of the user's categories, it does not use it directly. Instead, *httpd* passes ownership of the user's categories to the application server (Step 3), where application code uses it to communicate with the user data server (Step 4). *httpd* asks the exporter on its front-end machine to delegate its trust of these categories to the application server. To be considered valid, the delegation certificates created by the front end's exporter must be presented together with a chain of certificates up to the category's creator—the user data server—proving that the front-end machine was authorized to delegate trust in the first place. Since this chain includes the initial, short-lived certificate from the authentication proxy, malicious exporters cannot extend the amount of time they can act on the user's behalf, as long as the clock on the user data server does not go back.

## 5.5 Resource management

The distributed web server must also explicitly provide memory and CPU resources for all messages and processes. Since the *launcher* drives the execution of user requests, it requires such resources on all other machines. We use a special integrity category, $r_i$, to manage access to these resources, and each application and data server has an initial container labeled $\{r_i\}$, known to the *launcher*. The *launcher* owns $r_i$, giving it access to the initial container on those machines. We will describe later how this system is bootstrapped.

When the *launcher* starts *httpd*, it grants it ownership of $r_i$, and gives it the names of these initial containers on all other servers. When *httpd* talks to the authentication proxy in Step 1, for example, it uses the initial container on the corresponding user data server, along with its ownership of $r_i$, to send the request message.

Because HiStar labels all containers, the web server must take care to set the appropriate labels. Consider Step 4, when the application code wants to communicate with the file server. Although *httpd* could grant the application code ownership of $r_i$, the application code would not be able to use the initial container labeled $\{r_i\}$ on the data server, because the application code is labeled $\{u_s\}$ and cannot write to that container. Thus, *httpd* preallocates a sub-container with a label of $\{u_s, u_i\}$ on the user data server, using that machine's resource allocation service, and passes this container's name to the application code in Step 3. The application code can then use

this container to communicate with the file server, without being able to leak private user data through memory exhaustion.

Although in our prototype the application communicates with only one data server per user, a more complex application can make use of multiple data servers to handle a single request. Doing so would require the application code to have delegation certificates and access to containers on all of the data servers that it wants to contact. To do this, *httpd* could either pre-allocate all such delegations and containers ahead of time, or provide a callback interface to allocate containers and delegations on demand. In the latter case, *httpd* could rate-limit or batch requests to reduce covert channels.

### 5.6 Bootstrapping

When adding a new machine to our distributed web server, a bootstrapping mechanism is needed to gain access to the new machine's memory and CPU resources. For analogy, consider the process of adding a new machine to an existing Linux cluster. An administrator would install Linux, then from the console set a root password, configure an ssh server, and (if diligent about security) record the ssh host key to enter on other machines. From this point on, the administrator can access the new machine remotely, and copy over configuration files and application binaries. The ability to safely copy private data to the new machine stems from knowing its ssh host key, while the authority to access the machine in the first place stems from knowing its root password.

To add a new physical machine to a DStar cluster requires similar guarantees. Instead of an ssh host key, the administrator records the exporter's public key, but the function is the same, namely, for other machines to know they are talking to the new machine and not an impostor. However, DStar has no equivalent of the root password, and instead uses categories.

In fact, the root password serves two distinct purposes in Linux: it authorizes clients to allocate resources such as processes and memory on the new machine—i.e., to run programs—and it authorizes the programs that are run to access and modify data on the machine. DStar splits these privileges amongst different categories.

When first setting up the DStar cluster, the administrator creates an integrity category $r_i^1$ on the first HiStar machine (superscript indicates the machine), and a corresponding DStar category $r_i$ that we mentioned earlier. $r_i$ represents the ability to allocate and deallocate all memory and processes used by the distributed web server on any machine. It can be thought of as the "resource allocation root" for this particular application. However, there is no equivalent "data access root." Instead, different categories protect different pieces of data.

In configuring a new machine, the administrator's goal is to gain access to the machine's resources over the network, using ownership of $r_i$. The new machine's exporter initially creates a local category $r_i^n$ and a container labeled $\{r_i^n\}$ that will provide memory and CPU resources. To access this container, the administrator needs to establish a mapping between $r_i$ and $r_i^n$ on the new machine.

To do this, the administrator enters the name of $r_i$ when starting the exporter on the new machine, which then creates a mapping between $r_i$ and $r_i^n$. The exporter periodically broadcasts a signed message containing this mapping and the root container's ID, so the administrator need not manually transfer them.

With this mapping, processes that own $r_i$ can now copy files to the new machine and execute code there, much as the ssh client can on Linux if it knows the new machine's root password. However, a process that also owns other categories can use them to create files that cannot be read or written by owners of $r_i$ alone.

The current bootstrap procedure is tedious, requiring the manual transfer of category names and public keys. In the future, we envisage a setup utility that uses a password protocol like SRP [24] to achieve mutual authentication with an installation daemon, to automate the process. Alternatively, hardware attestation, such as TCPA, could be used to vouch that a given machine is running HiStar and a DStar exporter with a particular public key.

### 5.7 Replication

Having added a new machine to the DStar cluster, the administrator needs to securely replicate the web server onto it, and in particular, transfer the SSL private key to start a new *RSAd* process. We use a special replication daemon to do this, which ensures that the private key is only revealed to an *RSAd* binary on the remote machine.

To replicate *RSAd*, the administrator provides this daemon with a public key of the new machine, and access to a container on it (such as by granting it ownership of $r_i$). The replication daemon uses the mapping service to create a new category $rsa_s^n$ on the new machine, which will protect the private key there. To ensure that the private key is not passed to the wrong process, the replication daemon uses guarded invocation to invoke an authentic *RSAd* process on the new machine with ownership of $rsa_s^n$, and passes it the private key protected with $rsa_s^n$. Note that the administrator, who granted the replication daemon access to a container on the new machine, cannot read the private key that is now stored there, because he does not own $rsa_s^n$.

Both the replication daemon and the guarded invocation service, consisting of 120 and 200 lines of C++ code respectively, must be trusted to keep the private key secure, in addition to *RSAd*. A similar mechanism is used to start the *launcher*. HiStar's system-wide persistence

eliminates the need for a trusted process to start *RSAd* and *launcher* when the machine reboots.

### 5.8 Heterogeneous systems

To illustrate how DStar facilitates incremental deployment, we show how Linux can use HiStar or Flume to execute untrusted perl code with strong security guarantees. We implemented a DStar RPC server on HiStar and on Flume that takes the source code of a perl script and input data, executes the script on that input, and returns perl's exit code and output. DStar translates information flow restrictions specified by the caller into HiStar or Flume labels, which are then enforced by the operating system.

This service can be used by an existing Linux web server to safely execute untrusted perl code. The Linux machine can specify how different instances of perl can share data, by specifying the policy using secrecy and integrity categories in the request's label. To ensure each request is processed in complete isolation, fresh secrecy and integrity categories can be used for each request. If different scripts running on behalf of the same user need to share data, such as by storing it in the file system on a HiStar machine, the same categories should be used for each request made on behalf of a given user.

DStar also allows building distributed applications using a mix of operating systems, such as both Flume and HiStar. This may be useful when no single operating system is a perfect fit in terms of security or functionality for every part of the application. However, this paper does not evaluate any such applications.

## 6 EVALUATION

To quantify the overheads imposed by DStar, we present a detailed breakdown of DStar's data structure sizes and measure the cost of generating and verifying certificate signatures. To put these costs in perspective, we evaluate the performance of our web server and perl service under various conditions. The overheads introduced by both HiStar and DStar in the web server are acceptable for compute-intensive web applications such as PDF generation, and the performance is close to that of a Linux system. On the other hand, our web server delivers static content much more slowly than Linux.

Benchmarks were run on 2.4GHz Intel Xeon machines with 4MB CPU caches, 1GB of main memory, and 100Mbps switched Ethernet. For web server comparison, we used Apache 2.2.3 on 64-bit Ubuntu 7.04 with kernel version 2.6.20-15-generic. This Linux web server forked off a separate application process to handle every client request. The PDF workload used a2ps version 4.13b and ghostscript version 8.54. Xen experiments used Xen version 3.0 and kernel 2.6.19.4-xen for all domains. The ***netd*** TCP/IP stack was running a HiStar user-mode port of Linux kernel 2.6.20.4. Web servers used OpenSSL 0.9.8a with 1024-bit certificate keys; DStar exporters used 1280-bit Rabin keys.

| Data structure | Raw bytes | Compressed bytes |
|---|---|---|
| Public key | 172 | 183 |
| Category name | 184 | 195 |
| Category mapping | 208 | 219 |
| Unsigned delegation certificate | 548 | 384 |
| Signed delegation certificate | 720 | 556 |
| Unsigned address certificate | 200 | 203 |
| Signed address certificate | 376 | 379 |
| Null message (1) | 200 | 194 |
| Empty message (2) | 1348 | 623 |

**Figure 9**: Size of DStar data structures. Delegation certificate delegates a category to another exporter, and is signed by the category's creator. Null message (1) has an empty label and no mappings, delegations, or payload. Empty message (2) has a label consisting of one category, and includes one delegation certificate, one mapping, and an empty payload. The compressed column shows the potential reduction in size that can be achieved by compressing the data structures using zlib.

| Operation | Time (msec) |
|---|---|
| Sign a delegation certificate | 1.37 |
| Verify a delegation certificate | 0.012 |
| Sign an address certificate | 1.35 |
| Verify an address certificate | 0.011 |
| Null RPC on same machine | 1.84 |

**Figure 10**: Microbenchmarks measuring the time to sign and verify certificates, and the round-trip time to execute a null RPC request on one machine, with an empty label and no delegations or mappings.

### 6.1 Protocol analysis

Figure 9 shows the detailed sizes of DStar data structures as implemented in our prototype. The main source of space overhead in DStar messages is the public keys used to name categories and exporters. However, public keys are often repeated multiple times in the same message. For example, user secrecy and integrity categories are often created by the same exporter, and therefore share the same public key. Moreover, all of the delegations included in a message typically mention the same public key of the sending exporter. As a result, compressing messages results in significant space savings, as shown in the "compressed" column; however, our current prototype does not use compression. Storing only a hash of the public key in a category name can reduce its size, but would likely not reduce the size of a compressed message: delegation certificates in the same message are likely to include the entire public key (in order to verify certificate signatures), and with compression, there is little overhead for including a second copy of the same public key in the category name. Compressing the entire TCP session between two exporters, prior to encryption, is likely to generate further space savings, since the public keys of the two exporters are likely to be mentioned in each message. However, stream compression can lead to covert channels: the throughput observed by one process reveals the similarity between its messages and those sent by other processes.

| System | PDF workload | | cat workload | |
|---|---|---|---|---|
| | throughput | latency | throughput | latency |
| Linux Apache | 7.6 | 137 | 110.3 | 15.7 |
| HS PS+Auth | 6.3 | 161 | 37.8 | 30.8 |
| HS PS | 6.6 | 154 | 49.5 | 24.6 |
| HS no PS | 6.7 | 150 | 71.5 | 19.1 |
| DS on one machine | 5.2 | 194 | 17.0 | 63.0 |
| DS on multiple machines | varies | 511 | varies | 345 |

**Figure 11**: Maximum throughput (requests/sec) and minimum latency (msec) for the PDF and cat workloads on one machine. "HS PS+Auth" ran the HiStar web server from Section 5.1. "HS PS" ran the same web server without HiStar's user authentication service. "HS no PS" ran the same web server without privilege separation, with all components in a single address space. "DS" refers to the distributed web server.

The CPU overhead of public key operations on DStar certificates is shown in Figure 10. The Rabin cryptosystem used by DStar provides relatively fast signature verification but more expensive signature generation, which may be a reasonable trade-off for applications that do not change trust relations with each message. The overhead of issuing a local RPC, with an empty label and no mappings or delegations, is also shown in Figure 10.

## 6.2 Application performance

To evaluate the performance of applications running on our web server, we approximate a realistic service using a PDF generation workload. For each request, an HTTPS client connects to the server and supplies its user name and password. The web server generates a 2-page PDF document based on an 8KB user text file stored on the server, and sends it back to the client. Clients cycle through a small number of users; since the web server does not cache any authentication state, this does not skew performance. With multiple front-end servers, clients use a round-robin selection policy. We measure the minimum latency observed when a single client is active, and the maximum throughput achieved with the optimal number of clients; the optimal number of clients is generally proportional to the number of servers.

Figures 11 and 12 show the results of this experiment. A non-privilege-separated web server running on HiStar provides 12% less throughput than Linux; the difference is in part due to the overhead of fork and exec on HiStar. Privilege separation of the web server on HiStar imposes a 2% penalty, and HiStar's user authentication service, which requires eight gate calls, reduces throughput by another 5%. Running the distributed web server on a single machine shows the overhead imposed by DStar, although such a setup would not be used in practice. The throughput of the PDF workload scales well with the total number of servers.

## 6.3 Web server overhead

To better examine the overhead of our web server, we replaced the CPU-intensive PDF workload with `cat`, which just outputs the same 8KB user file; the results are also shown in Figures 11 and 12. Apache has much higher throughput than the HiStar web server both with and without privilege separation. Though Apache serves static content better without using `cat`, we wanted to measure the overhead of executing application code. Our web server's lower performance reflects that its design is geared towards isolation of complex application code; running simple applications incurs prohibitively high overhead. Nonetheless, the distributed web server still scales with the number of physical machines.

| Calling machine | Linux | HiStar | Linux | Linux | Linux |
|---|---|---|---|---|---|
| Execution machine | same | same | Linux | HiStar | Flume |
| Communication | none | none | TCP | DStar | DStar |
| Throughput, req/sec | 505 | 334 | 160 | 67 | 61 |
| Latency, msec | 2.0 | 3.0 | 6.3 | 15.7 | 20.6 |

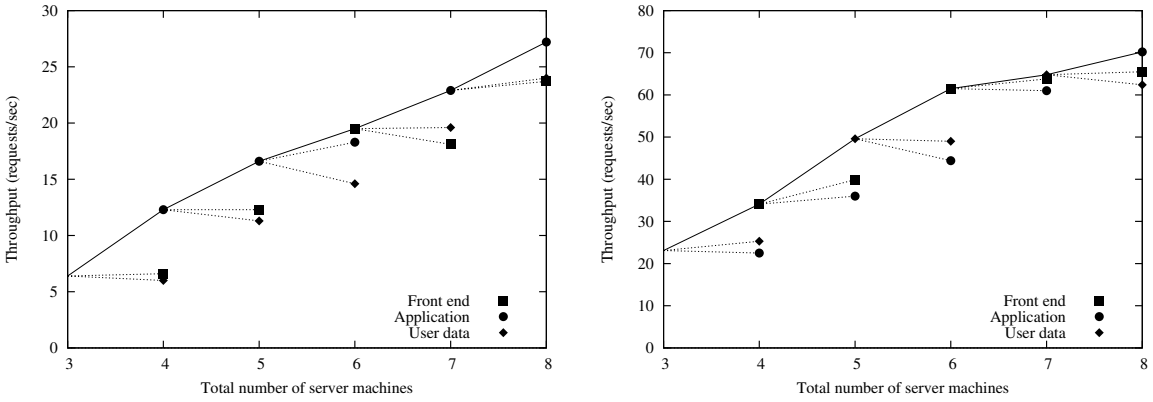
**Figure 13**: Throughput and latency of executing a "Hello world" perl script in different configurations.

## 6.4 Privilege separation on Linux

Is it possible to construct a simpler, faster privilege-separated web server on Linux that offers similar security properties? We constructed a prototype, running separate *launcher*, *SSLd*, *RSAd*, and Apache processes, using chroot and setuid to isolate different components from each other. As in the earlier Apache evaluation, a fresh application process was forked off to handle each client request. This configuration performed similar to a monolithic Apache server. However, to isolate different user's application code from one another, Apache (a 300,000 line program) needs access to setuid, and needs to run as root, a step back in security. We can fix this by running Apache and application code in a Xen VM; this reduces the throughput of the PDF workload to 4.7 req/sec. Even this configuration cannot guarantee that malicious application code cannot disclose user data; doing so would require one VM per request, a fairly expensive proposition. This suggests that the complexity and overhead of HiStar's web server may be reasonable for the security it provides, especially in a distributed setting.

## 6.5 Heterogeneous systems

Figure 13 shows the latency and throughput of running a simple "untrusted" perl script that prints "Hello world" on Linux, on HiStar, on Linux invoked remotely using a simple TCP server, and on HiStar and Flume invoked remotely using DStar from Linux. A fresh secrecy category is used for each request in the last two cases. This simple perl script provides a worst-case scenario, incurring all of the overhead of perl for little computation; more complex scripts would fare much better. The lower perl performance on HiStar is due to the overhead of emulating fork and exec. DStar incurs a number of round-trips to allocate a secrecy category and create a container for ephemeral call state, which contributes to a significantly higher latency. Comparing the throughput on HiStar run-




(a) Throughput under the PDF workload

(b) Throughput under the `cat` workload

**Figure 12**: Maximum throughput achieved by the DStar web server running on multiple machines. The initial point on the left represents three machines: one front-end server, one application server, and one user data server. Subsequent points reflect the total throughput with an extra front-end server, application server, or user data server added, as indicated by the point symbol. The process is repeated for the highest-performing configuration at each step. The best use of an additional server machine is indicated by a solid line, and others by a dashed line.

ning locally and remotely shows that DStar adds an overhead of 12 msec of CPU time per request, which may be an acceptable price for executing arbitrary perl code from a Linux machine with well-defined security properties.

## 7 Related work

Flume [12] controls information flow in a fully-trusted centralized cluster sharing an NFS server and tag registry. Flume does not allow applications to directly communicate between machines, define their own trust relations, or share data across administrative domains. On the other hand, applications in DStar have complete control over trust relations for their data, and can communicate between any machines that speak the DStar protocol. Flume's centralized design limits scalability to small, fully-trusted local clusters, and cannot withstand any machine compromises. DStar could be used to connect multiple Flume clusters together without any inherent centralized trust or scalability bottlenecks.

Capability-based operating systems, such as KeyKOS [4] and EROS [17], can provide strict program isolation on a single machine. A DStar exporter could control information flow on a capability-based operating system by ensuring that processes with different labels had no shared capabilities other than the exporter itself, and therefore could not communicate without the exporter's consent.

Shamon [14] is a distributed mandatory access control system that controls information flow between virtual machines using a shared reference monitor. DStar avoids any centralized reference monitor for security and scalability. Shamon tracks information flow at the granularity of *x*86 virtual machines, making it impractical to track each user's data. DStar running on HiStar can apply policies to fine-grained objects such as files or threads.

A number of systems, including Taos [23] and Amoeba [19], enforce discretionary access control in a distributed system, often using certificates [3]. None of them can control information flow, as a malicious program can always synthesize capabilities or certificates to contact a colluding server. The Taos *speaks-for* relation inspired the much simpler DStar *trusts* relation, used to define discretionary privileges for different categories between exporters.

Multi-level secure networks [2, 9, 15, 18] enforce information flow control in a trusted network, but provide very coarse-grained trust partitioning. By comparison, DStar functions even in an untrusted network such as the Internet, at the cost of introducing some inherent covert channels, and allows fine-grained trust to be explicitly configured between hosts. Using a secure, trusted network would reduce covert channels introduced by DStar.

Unlike multi-level secure networks, DStar does not allow labeling a machine without giving it ownership privileges. Providing a non-empty machine label would require a trusted component to act as a proxy for the machine, ensuring that any packets sent or received by the machine are consistent with its current label. This can be done either with support from the network, or by explicitly forwarding messages through a proxy trusted to maintain the labels of machines it is proxying.

Secure program partitioning [25] partitions a single program into sub-programs that run on a set of machines specified at compile time with varying trust, to uphold an overall information flow policy. DStar is complementary, providing mechanisms to enforce an overall information flow policy without restricting program structure, language, or partitioning mechanism. DStar could execute secure program partitioning's sub-programs in a distributed system without trusting the partitioning com-

piler. Secure program partitioning has a much larger TCB, and relies on trusted external inputs to avoid a number of difficult issues addressed by DStar, such as determining when it is safe to connect to a given host at runtime, when it is safe to allocate resources like memory, and bootstrapping.

Jif [16] provides decentralized information flow control in a Java-like language. Although its label model differs from DStar's, a subset of Jif labels can be expressed by DStar. DStar could provide more fine-grained information flow tracking by enforcing it with a programming language like Jif rather than with an operating system.

Jaeger et al [11] and KDLM [7] associate encryption keys with SELinux and Jif labels, respectively, and exchange local security mechanisms for encryption, much like DStar. These approaches assume the presence of an external mechanism to bootstrap the system, establish trust, and define mappings between keys and labels—difficult problems that are addressed by DStar. Moreover, these approaches configure relatively static policies and trust relations at compile time; DStar allows any application to define new policies and trust at runtime. DStar transfers the entire label in each message, instead of associating labels with keys, since applications never handle ciphertexts directly. An application that receives a ciphertext but not the corresponding key may still infer confidential information from the size of the ciphertext.

## 8 Summary

DStar is a framework for securing distributed systems by specifying end-to-end information flow constraints. DStar leverages the security label mechanisms of DIFC-based operating systems. Each machine runs an exporter daemon, which translates between local OS labels and globally-meaningful DStar labels. Exporters ensure it is safe to communicate with other machines, but applications define the trust relations. Self-certifying categories avoid the need for any fully-trusted machines. Using DStar, we built a highly privilege-separated, three-tiered web server in which no components are fully-trusted and most components cannot compromise private user data even by acting maliciously. The web server scales well with the number of physical machines.

## Acknowledgments

We thank Michael Walfish, Siddhartha Annapureddy, Pavel Brod, Antonio Nicolosi, Junfeng Yang, Alex Yip, the anonymous reviewers, and our shepherd, Ken Birman, for their feedback. This work was funded by NSF Cybertrust award CNS-0716806, by joint NSF Cybertrust and DARPA grant CNS-0430425, by the DARPA Application Communities (AC) program as part of the VERNIER project at Stanford and SRI International, and by a gift from Lightspeed Venture Partners.

## References

[1] O. Acıiçmez, Çetin Kaya Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. Cryptology ePrint Archive, Report 2006/351, 2006. `http://eprint.iacr.org/`.

[2] J. P. Anderson. A unification of computer and network security concepts. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 77–87, Oakland, CA, 1985.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996.

[4] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[5] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proc. of the 12th USENIX Security Symposium*, August 2003.

[6] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. of the 15th USENIX Security Symposium*, Vancouver, BC, 2006.

[7] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *16th IEEE Computer Security Foundations Workshop*, pages 170–186. IEEE Computer Society, 2003.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[9] D. Estrin. Non-discretionary controls for inter-organization networks. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 56–61, Oakland, CA, 1985.

[10] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proc. of the 2002 Ottawa Linux Symposium*, pages 479–495, June 2002.

[11] T. Jaeger, K. Butler, D. H. King, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for mandatory access control across systems. In *Proc. of the 2nd International Conference on Security and Privacy in Communication Networks*, August 2006.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st SOSP*, Stevenson, WA, October 2007.

[13] D. Mazières. A toolkit for user-level file systems. In *Proc. of the 2001 USENIX*, pages 261–274, June 2001.

[14] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamon: A system for distributed mandatory access control. In *Proc. of the 22nd Annual Computer Security Applications Conference*, Washington, DC, USA, 2006. IEEE Computer Society.

[15] J. McHugh and A. P. Moore. A security policy and formal top-level specification for a multi-level secure local area network. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 34–39, Oakland, CA, 1986.

[16] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOCS*, 9(4):410–442, October 2000.

[17] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. of the 17th SOSP*, December 1999.

[18] D. P. Sidhu and M. Gasser. A multilevel secure local area network. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 137–143, Oakland, CA, 1982.

[19] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33:46–63, December 1990.

[20] Think Computer Corporation. Identity crisis. `http://www.thinkcomputer.com/corporate/whitepapers/identitycrisis.pdf`.

[21] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM TOCS*, 25(4):11:1–43, December 2007.

[22] C. Weissman. Security controls in the ADEPT-50 time-sharing system. In *Proc. of the 35th AFIPS Conference*, pages 119–133, 1969.

[23] E. P. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM TOCS*, 12(1):3–32, 1994.

[24] T. Wu. The secure remote password protocol. In *Proc. of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[25] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. of the 18th SOSP*, pages 1–14, October 2001.

[26] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, November 2006.